SCL: A Simple and Efficient Solution for Deterministic Computation using Bitcoin

Paul Dando
paul@darkfusion.tech

Ashif Asharaph
ashif@darkfusion.tech

August 5, 2026

Abstract

This paper presents the Simple Contract Language (SCL), a decentralized framework for lightweight and deterministic computation in permissionless environments. SCL builds upon the Bitcoin transaction model to achieve three core objectives: (i) immutably anchoring authorization on Bitcoin, both for contract deployment and for every subsequent call; (ii) introducing deterministic execution contexts; and (iii) providing consensus guarantees alongside structural resilience against common attack vectors, including replay, front-running, and data-withholding attacks. By aligning computation with Bitcoin's proven security model, SCL enables efficient, verifiable, and scalable contract execution without the overhead of traditional on-chain virtual machines, and extends naturally to instant off-chain asset transfers over the Lightning Network.

1 Introduction

1.1 Background

The concept of smart contracts and the vision of a decentralized supercomputer have been central to the evolution of blockchain technology, inspiring much of the industry as it exists today. While widely adopted and mature in certain applications, the prevailing model of on-chain computation is fundamentally limited. Executing arbitrary computation directly on-chain has proven costly, inefficient, and difficult to scale, creating barriers to real-world deployment and global adoption. These constraints highlight the need for innovation beyond the current paradigm. To unlock the next generation of blockchain applications, a shift is required: one that decouples computation from costly consensus mechanisms while preserving security, trust, and verifiability.

1.2 The Paradigm Shift

A shift in approaches to smart contract execution is already underway. The concept of *client-side validation*, first proposed by Todd in 2016 [2], and later refined through the work of Orlovsky and the RGB project [3], exemplifies the industry's search for alternatives to conventional on-chain computation. These efforts underscore the pressing need for innovation, yet they also illustrate inherent trade-offs: while client-side validation mitigates some of the inefficiencies of on-chain execution, it often does so at the expense of decentralization, usability, or system complexity. In contrast, *Simple Contract Language (SCL)* offers a streamlined design that prioritizes both security and scalability, while minimizing the compromises typically associated with alternative approaches.

2 System Overview

SCL provides a deterministic execution model with a single, public source of truth: Bitcoin. Contracts are compiled into bytecode C, and a cryptographic commitment $h = \mathrm{H}(C)$ is immutably embedded into a Bitcoin UTXO. Every subsequent call is itself anchored: the call travels as a signed *call envelope*, and a commitment to that envelope is carried on Bitcoin in a *protocol stamp* inside the anchoring transaction (Section 6). State transitions are validated off-chain by replaying payloads against C and Bitcoin-derived context (e.g., UTXO set, block height, signatures), and each node folds the resulting write-sets into a per-block *state root*. This anchoring ensures immutability, a canonical total ordering of all state transitions by Bitcoin block, and objective validation.

3 Architecture

Execution Layer. Nodes execute contracts off-chain in the SCL VM. The VM is deterministic, bounded, and non–Turing-complete, ensuring replayable state transitions.

Settlement Layer. Compiled contracts C are anchored via cryptographic commitment $h = \mathrm{H}(C)$ stored in Bitcoin, and every confirmed call is ordered and finalized by the Bitcoin transaction that carries its protocol stamp. This provides immutability, global verifiability, and a canonical ordering of all state transitions.

Validator Network. Validators independently replay payloads in a two-phase pipeline (intake and confirmation, Section 6) and publish per-block state roots. There is no consensus ceremony: honest nodes reach the same verdict by verifying consistency with h, the protocol stamp, and Bitcoin-derived context.

4 Execution Model and Language

4.1 Determinism and Boundedness

SCL deliberately forbids nondeterministic constructs such as time, randomness, or external oracles. This ensures that for any given payload $p \in \mathcal{P}$ and execution context $\Gamma \in \mathcal{C}$, the state transition Δ is uniquely defined by a total deterministic function:

$$\Delta = f_C(p, \Gamma), \quad f_C : \mathcal{P} \times \mathcal{C} \to \mathcal{S},$$

where $\mathcal{S}$ denotes the state space. Determinism implies that two independent evaluators, given identical inputs, will produce identical state deltas:

$$f_C(p, \Gamma) = f_C(p, \Gamma) \quad \forall\, p \in \mathcal{P}, \Gamma \in \mathcal{C}.$$

To mitigate denial-of-service and ensure computational predictability, SCL enforces *hard ceilings* on execution cost. Each execution is constrained by an instruction budget

$$N \leq 10^5,$$

stack depth

$$d \leq 1024,$$

and call depth

$$d_{\text{call}} \leq 64,$$

with persistent storage bounded at 64 KB per contract and payloads capped at 64 KB. The instruction budget is shared across nested cross-contract calls: however deep a call chain goes, the entire transaction runs under a single budget, so composition cannot amplify cost. Formally, for an execution trace $\tau = (s_0, s_1, \ldots, s_n)$, where each s_i is a machine state and $n = |\tau|$, SCL enforces the invariant

$$n \leq N \quad \wedge \quad \max_i \text{depth}(s_i) \leq d.$$

These invariants guarantee termination within polynomially bounded resource usage. Since each transition step can be evaluated in time $O(1)$ relative to the instruction budget, the total cost of evaluating a payload is bounded by

$$T(p, \Gamma) \in O(N).$$

Thus, every execution of SCL resides within $\mathbf{PTIME}$, and more specifically within $\mathbf{O}(10^5)$ steps, regardless of the adversarial structure of p. This combination of determinism and bounded execution ensures that safety properties (e.g., totality, termination) are provable, and denial-of-service vectors such as infinite loops, unbounded recursion, or uncontrolled memory allocation are mathematically excluded from the operational semantics.

4.2 Language Features

In SCL, each contract is authored in its own file and comprised of four essential components:

- **Constants** declared via `const`, used for embedding immutable values that remain fixed at compile time.
- **Fields**, which store the mutable state and are annotated with access-control via attributes like `@writeable_by(init, bump)`. Such annotations are enforced at code generation time; if a function not authorized attempts to modify a protected field, the assignment is silently skipped without causing a runtime error.
- **Events**, declared at contract level via `event` and raised with `emit`, giving contracts a typed, queryable record of what happened.
- **Functions** marked by the `fn` keyword, which deterministically evolve the contract's state through state-modifying logic that respects write constraints and deterministic semantics.

A minimal illustrative contract from the documentation:

```
contract Minimal {
    // constants
    const STEP: u64 = 1;

    // fields
    @writeable_by(init, bump)
    counter: u64;

    // events
```

```
    event Bumped(value: u64);

    // functions
    fn init(x: u64) {
        counter = x;
    }

    fn bump() {
        counter = counter + STEP;
        emit Bumped(counter);
    }
}
```

Supported Types. SCL supports a restrained but expressive type system, allowing for rich yet well-typed contract logic:

- **Scalar types**: `u64`, `u8`, `bool`, `Str`, `String`.
- **Collections**: `List⟨T⟩` for homogeneous sequences, `HashMap⟨K, V⟩` for key-value mappings, and tuples written as `(T1, T2, ...)`.
- **Literals**: numeric literals, strings, byte literals such as `b"..."` or hex-formatted, list literals like `[1, 2]`, and map literals like `{"k": 1}`.

Notably, boolean values are compiled into integers at runtime: `true` becomes 1 and `false` becomes 0. Branching on booleans requires explicit comparison against 0 or 1.

Control Flow Constructs. SCL enables classic structured control flow, with syntax analogous to mainstream languages:

- Standard conditional branching using `if`, `else if`, and `else`. Example:

```
fn guard(n: u64) {
    if (n == 0) {
        PRINT("zero");
    } else {
        PRINT("nonzero");
    }
}
```

- Looping via `for` loops over concrete collections. Locals are immutable unless declared `let mut`. Example:

```
fn total(entries: List) {
    let mut s = 0;
    for e in entries {
        s = s + e;
    }
    PRINT("sum:", s);
}
```

  There is no `while` loop and no unbounded iteration: every loop runs over a concrete collection, so every loop is bounded by construction. Recursion is syntactically possible but capped hard by the VM's 64-frame call-depth limit.

Events. Events are collected during execution, including those emitted by nested cross-contract calls, and are recorded when the call confirms on Bitcoin. Each stored event carries the Bitcoin transaction identifier of the emitting call and its confirmation height, and event schemas appear in the contract ABI, typed like function inputs. Nodes expose per-contract event queries filtered by event name and block range, so indexers and frontends can build provable activity feeds without contract-specific glue: state answers "what is," events answer "what happened," and both are provable against Bitcoin.

4.3 Built-ins and Context

The SCL runtime exposes a carefully curated set of *built-in functions*, designed to balance expressiveness with safety and determinism. These built-ins provide access to persistent state, debugging aids, blockchain context, and controlled inter-contract communication. By restricting developers to this finite and well-defined set of primitives, SCL ensures that all contract executions remain deterministic and resource-bounded.

Persistent Storage. Contracts are endowed with persistent key–value collections, exposed via map and set operations. These enable contracts to maintain state across invocations, while bounding storage operations within deterministic complexity classes:

$$\text{MAP\_SET}(k, v) : (K \times V) \to \top, \quad \text{MAP\_GET}(k) : K \to V \cup \{\bot\}, \quad \text{MAP\_REMOVE}(k) : K \to \top,$$

$$\text{SET\_ADD}(x) : T \to \top, \quad \text{SET\_CONTAINS}(x) : T \to \{0, 1\}, \quad \text{SET\_REMOVE}(x) : T \to \top.$$

All per-entry map and set operations execute in $O(1)$ expected time, ensuring contract performance is insensitive to the size of stored data. Whole-map reads (MAP_GET_ALL) and bounded list accessors (LIST_LEN, LIST_GET) round out the collection primitives; all remain subject to the shared instruction budget.

Debugging and Introspection. For development and verification, SCL provides:

$$\text{PRINT}(x) : T \to \top, \quad \text{TYPEOF}(x) : T \to \text{Str}.$$

These functions are side-effect free with respect to consensus-critical state; their effects are visible only during contract development or via log outputs.

Blockchain Context. Contracts can introspect Bitcoin-level execution state via dedicated context functions:

$$\text{SENDERS}() \to \text{List[PubKey]}, \quad \text{RECEIVERS}() \to \text{List[PubKey]}, \quad \text{SENDER\_ADDRESS}() \to \text{PubKey},$$

$$\text{CONTRACT\_ID}() \to \text{ContractID}, \quad \text{block\_height}() \to \mathbb{N}.$$

Here, `SENDERS()` returns the set of public keys that authorized the transaction inputs (UTXO spenders), while `RECEIVERS()` exposes the public keys of transaction outputs; `SENDER_ADDRESS()` is the standard way to gate functions on the caller, and `CONTRACT_ID()` lets a contract refer to itself. This binding of contract execution to the Bitcoin UTXO model ensures that contract logic is always grounded in the consensus state of the underlying chain.

Cross-contract Communication. To enable modular design, SCL supports controlled invocation of other contracts:

$$\text{CALL}(c, p) : (\text{ContractID} \times \mathcal{P}) \to \Delta,$$

where c is the target contract identifier, p a payload, and Δ the resulting state delta. `CALL` propagates the caller's identity into the callee; its companion `CALL_AS_CONTRACT` invokes the callee with the calling *contract* as the sender, the standard pattern for escrow and staking contracts that hold balances in their own name. Nested calls share a single instruction budget and the global ceilings ($N \leq 10^5$, stack depth ≤ 1024, call depth ≤ 64), ensuring that cross-contract execution cannot be abused for denial-of-service or cost amplification. Events emitted by callees are collected alongside the caller's.

Transfers and Signature Verification.

$$\text{TRANSFER}(S, R, A) : (\text{List} \times \text{List} \times \text{List}) \to \{0, 1\}, \quad \text{CHECK\_SIG}(\sigma, m, pk) : \text{Bytes}^3 \to \{0, 1\}.$$

TRANSFER performs a balance-checked multi-party transfer whose sender and receiver totals must reconcile, and CHECK_SIG deterministically verifies ed25519 and secp256k1 signatures, allowing contracts to validate off-chain authorizations in-band.

4.4 Turing Completeness Trade-offs

A central critique of many lightweight or alternative smart contract frameworks is the absence of *Turing completeness*, which theoretically allows arbitrary computation. However, practical Turing-complete blockchain environments have consistently shown high execution costs, inefficiency, and state bloat.

In designing SCL, we deliberately forgo strict Turing completeness to achieve **determinism, efficiency, and verifiability**. This trade-off is negligible in practice: most real-world contracts (ie. conditional transfers, state commitments, proofs of authorization, or structured data exchanges) do not require unrestricted computation. Instead, they demand predictable execution, low cost, and scalable verification.

Restricting SCL to deterministic and bounded operations ensures that execution is efficient and reliable across all nodes, mitigating the risk of runaway computation or adversarial exploits. While not Turing complete in theory, the language remains sufficiently expressive for the overwhelming majority of practical applications.

5 Compiler, Virtual Machine, and Node Architecture

This section describes the concrete implementation architecture underlying SCL:

- the SCL compiler, which transforms human-readable contracts into bytecode;
- the SCL virtual machine (VM), which executes that bytecode deterministically; and
- the SCL node, which embeds the VM into a networked validator and settlement stack.

In principle, the design is *chain-agnostic*. The language and execution model do not assume any specific Layer 1 implementation beyond a UTxO-style transaction model, while the current reference node implementation targets Bitcoin as its settlement layer.[1]

5.1 SCL Stack Overview

At a high level, SCL contracts flow through the stack as follows:

1. A developer writes a contract in SCL.
2. The SCL compiler parses, validates, and compiles this source into a compact bytecode C together with associated metadata (constant pool, function offsets, variable layout).
3. The SCL VM executes bytecode C in a sandboxed, deterministic environment, using:
   - a local, contract-scoped state store,
   - a fixed set of built-ins, and
   - Bitcoin-derived context (UTXOs, block height, transaction senders/receivers).
4. An SCL node orchestrates compilation artifacts, VM execution, networking, and persistence, and anchors contract commitments on Bitcoin.

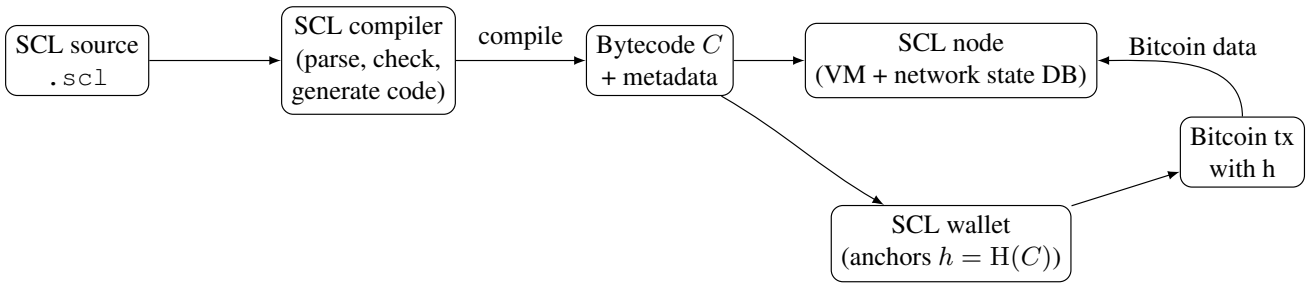

Figure 1: End-to-end path from SCL source code to compiled bytecode and execution. The SCL wallet typically computes the hash $h = \mathrm{H}(C)$ and anchors it in a Bitcoin transaction. SCL nodes then observe the Bitcoin chain, use the anchored commitment and relevant UTXOs as context, and execute the bytecode in the VM over the contract state to validate and update SCL contract state.

5.2 Compiler Design and Tooling

The SCL compiler is implemented as a modular pipeline with clearly separated concerns. Its role is to map a contract written in SCL to a canonical bytecode representation C and associated metadata used by the VM and by other tools in the SCL ecosystem (wallets, deployment scripts, etc.).

Front-end: Parsing and Static Analysis. The front-end performs:

- **Lexical analysis**, transforming the source text into a token stream.
- **Parsing**, constructing an abstract syntax tree (AST) for contracts, fields, functions, expressions, and statements.
- **Symbol table construction**, tracking declared constants, fields, locals, built-ins, and their scopes.
- **Semantic checks** that enforce core language constraints:
  - rejection of syntactically unsupported control-flow constructs (only bounded `for`-style loops are admitted by the grammar);

[1] At the time of writing, the production implementation focuses on Bitcoin Layer 1; support for additional UTxO-based layers and chains is not assumed by this paper.

- enforcement of field access rules (e.g., `@writeable_by` annotations are honored during code generation so that unauthorized writes are not emitted);
- validation of builtin usage and argument counts/types against a fixed registry of supported built-ins.

Contracts that violate these constraints fail to compile. Together with the VM's instruction, stack, and call-depth limits, this ensures that only contracts satisfying the determinism and boundedness guarantees of Section 4 can be deployed.

Intermediate Representation and Code Generation. After successful analysis, the compiler lowers the AST into a small instruction set tailored to SCL's stack-based VM. This intermediate representation is then assembled into:

- a **constant pool**, storing literals and static data;
- a **bytecode stream** C encoding VM instructions and jump targets;
- a **function offset table**, mapping function identifiers to byte offsets in C;
- **per-function local layout metadata**, such as the number of local variables and their allocation in the VM stack frame.

The compiler exposes this structured metadata that tooling can use, together with the original source, as well as an ABI of a contract's public interface. The same compilation pipeline is exposed both:

- as a command-line tool for local development and continuous integration; and
- by the hosted SCL IDE and online compiler.

This ensures that contracts compiled in-browser and contracts compiled offline share identical semantics.

Commitment and Canonicalization. The canonical bytecode C from the compiled artifact is hashed and embedded into Bitcoin. Any change to the source that affects semantics yields a different C and thus a different h as detailed in Section 9. This binding allows independent nodes to verify that the bytecode they execute corresponds to the on-chain commitment without needing access to the original source.

5.3 Virtual Machine Execution Model

The SCL VM is a deterministic, stack-based interpreter specialized for the bytecode generated by the compiler. It is designed to be:

- **sandboxed**: no direct access to system time, randomness, or the network;
- **resource bounded**: each execution is constrained by instruction count, stack depth, call depth, and storage limits; and
- **chain-aware**: Bitcoin and UTxO information is supplied as read-only context.

Each VM instance is parameterized by:

- the compiled bytecode C and constant pool;
- an array of local variables and contract fields;
- a stack for intermediate values;
- an optional handle to a persistent key–value store that backs contract fields and collections;
- a context map of named values (e.g., block height, transaction identifiers, public keys of senders and recipients);
- configurable limits (maximum instructions, maximum stack size, maximum call depth, maximum storage per contract).

Given a payload p and context Γ, the node constructs the initial VM state and runs the bytecode until one of the following occurs:

1. the program terminates successfully (normal return);
2. the VM reaches a resource limit (instruction budget, stack depth, call depth, or storage bound); or
3. a deterministic error condition is encountered (e.g., invalid opcode, type mismatch in a built-in).

In all cases, the outcome is fully determined by (C, p, Γ) and the prior contract state, with no hidden sources of nondeterminism.

Persistent Storage and Context. The VM exposes the storage and context facilities described in Section 4 as built-ins:

- map and set operations backed by a contract-scoped key–value store; and
- read-only accessors for Bitcoin-derived data (senders, receivers, block height, etc.).

In the reference implementation, persistent storage is powered by an embedded key–value database, and contract state is logically namespaced by a unique contract identifier. In principle, the VM itself is agnostic to the concrete storage engine; it requires only a deterministic key–value interface.

Cross-contract Calls. SCL supports bounded cross-contract calls via the `CALL` built-in and a global contract registry maintained by the node. Each registered contract is associated with its bytecode, constants, function offset metadata, and access to the shared persistent storage backend (namespaced by contract identifier). When a contract invokes another via `CALL`:

1. the VM resolves the target contract and function in the registry;
2. it instantiates a fresh VM interpreter for the callee contract, reusing the same underlying storage handle but isolating execution state (stack, locals, instruction pointer);
3. it executes the callee function under the caller's *remaining* resource budget: nested calls share one instruction budget and the global stack and call-depth ceilings, so composition cannot amplify cost; and
4. it returns the callee's explicit return value (a typed VM value) to the caller. If the call fails or the callee returns no value, the reference implementation returns `0` as an error sentinel.

This design allows modular composition while preserving deterministic replay, with failures yielding a deterministic result.

5.4 Node Integration and Network Role

The SCL node embeds the compiler and VM into a full networked application that:

- ingests SCL-related payloads;
- derives Bitcoin context from a configured backend;
- executes contract logic in the VM; and
- persists contract state, payload history, and auxiliary indexes for explorers and wallets.

Conceptually, the node is composed of the following subsystems:

API layer. Exposes REST and gRPC endpoints to deploy contracts, submit signed call envelopes, poll call status, read contract state and per-block state roots, and query per-contract events by name and block range.

Consensus and execution layer. Normalizes incoming payloads into a canonical internal format, extracts the referenced contract identifier and function, and invokes the VM. Domain-specific logic (e.g., list/bid/transfer flows) is implemented by SCL contracts; the node coordinates their evaluation but does not introduce an independent consensus protocol.

Storage layer. Maintains:

- queues of pending and confirmed payloads,
- the set of known contracts and their compiled artifacts,
- cached Bitcoin transaction information and UTXO state, and
- auxiliary data structures (e.g., indexes, validator presence records).

Network and synchronization. Handles peer discovery and synchronization of contract state and payload history. To minimize bandwidth, the node can exchange compact summaries (e.g., Merkle roots and diffs) rather than full histories.

Validator coordination (optional). For networks that reward validation, an optional coordinator/validator mode uses cryptographic challenges and signed presence claims to measure validator participation over time. Importantly, this mechanism does not alter the fact that disputes about contract execution are objectively resolvable by replaying bytecode against Bitcoin-derived context.

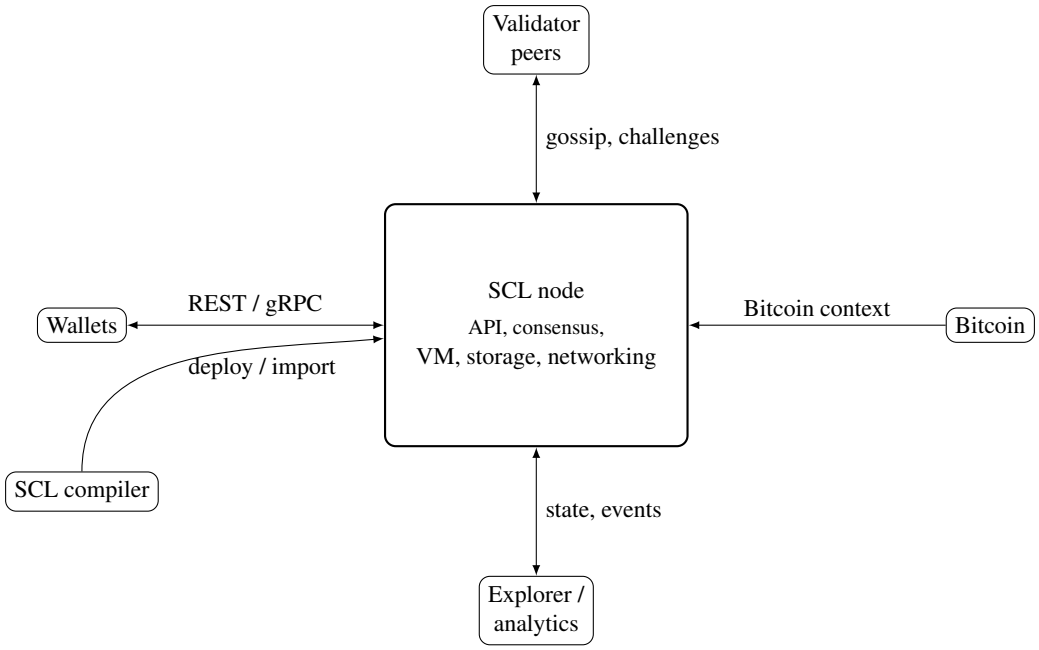

Figure 2: The SCL node within the broader SCL stack: compiler, wallets, explorer, validator network, and Bitcoin.

5.5 Developer Workflow: From Source Code to Execution

From a developer's perspective, the path from SCL source code to live execution proceeds through the following steps:

1. **Authoring.** The developer writes an SCL contract file, defining constants, fields, and functions according to the language model in Section 4. This can be done in a local editor or in the hosted SCL IDE.

2. **Compilation.** The contract is compiled using either:

   - the command-line compiler; or
   - the web-based compiler backed by the same compilation pipeline.

   The compiler outputs bytecode C, metadata (constant pool, function offsets, field layout), and an ABI description.

3. **Anchoring on Bitcoin.** A deployment tool or wallet constructs a Bitcoin transaction that embeds $h = \mathrm{H}(C)$ (for example, in an `OP_RETURN` output or other commitment-compatible pattern). This transaction is broadcast and settled on Bitcoin, providing an immutable anchor for the contract's semantics. Every subsequent *call* to the contract is anchored the same way, via a protocol stamp committing to its call envelope (Section 6).

4. **Registration in the Node.** Upon receiving a deployment (or invocation) request, the node persists the submitted artifact and request data keyed by the associated Bitcoin transaction identifier (txid) and marks it as pending. The node monitors the Bitcoin chain and, once the txid reaches a configurable confirmation threshold, promotes the entry to active and registers it in the in-memory contract registry (using the same txid as the contract identifier in the reference implementation).

5. **Invocation and Validation.** To invoke a contract function, a client submits a signed *call envelope* that carries:

   - the network and contract identifiers,
   - the function name and serialized arguments,
   - the caller's public key, a unique call identifier, and the caller's signature, and
   - the Bitcoin outpoint that the call's anchoring transaction must spend.

   The node:

   (a) validates the envelope's anchor binding at intake and queues the pending call;

   (b) once the anchoring transaction confirms, re-validates the binding against the on-chain protocol stamp, resolves the commitment h, and fetches relevant Bitcoin context (UTXOs, block data);

   (c) constructs the VM context Γ and executes $f_C(p, \Gamma)$ in the SCL VM;

   (d) persists the resulting state delta with undo data, folds it into the block's state root; and

(e) returns a structured result (e.g., success/failure, logs, emitted events, updated state summary) to the client.

This workflow is designed so that each step is independently verifiable. Any third party with access to the same inputs (contract bytecode C, payload p, context Γ derived from Bitcoin, and prior state) can reconstruct and validate the node's verdict. It is expected that new nodes joining the network typically traverse this process, replaying contract history from the same public inputs, to independently validate the current contract state and satisfy themselves as to its correctness.

6 Per-Call Anchoring and Settlement

Anchoring in SCL is not limited to deployment: every contract call is committed on Bitcoin. This section describes the call container, the on-chain commitment, and the two-phase validation pipeline that together make replay, re-anchoring, and front-running structurally impossible.

6.1 Call Envelopes and Protocol Stamps

A call travels as a signed *call envelope*: the network identifier, contract identifier, function name, serialized arguments, caller public key, a unique call identifier, and the caller's signature. Crucially, the envelope also names the exact Bitcoin outpoint that its anchoring transaction must spend.

The anchoring transaction carries a *protocol stamp* in an `OP_RETURN` output: a compact commitment to the envelope hash and the contract identifier. When the transaction confirms, the call is final and totally ordered by its position in the Bitcoin chain.

The anchor-outpoint binding closes replay and front-running holes structurally rather than heuristically:

- an envelope cannot be re-anchored in a different transaction, because the anchor outpoint would not match;
- a third party cannot front-run a call with their own transaction, because they cannot spend the caller's outpoint; and
- the confirmed stamp binds the routed function name to the committed envelope, so a forged wrapper cannot redirect a call to a different function than the one committed on-chain.

6.2 Two-Phase Validation

Nodes validate every call twice. **At intake** (from a client or gossip), the node parses the envelope, validates the anchor binding (the anchoring transaction's first input must spend the outpoint the envelope names), and queues the pending call. **At confirmation** (block by block), the node scans each Bitcoin block for protocol stamps and, for each one found: re-validates the anchor binding, a consensus-critical check under which a mismatched or unparseable envelope is invalid on every honest node with no execution; derives context Γ from Bitcoin (block height, sender address, and the anchor transaction's inputs and outputs); deterministically executes the call against the anchored bytecode; and commits the resulting write-set together with undo data. Invalid payloads fail locally; no global voting is required.

6.3 State Roots and Replay Consistency

At every Bitcoin block, each node computes a Merkle *state root* over all contract state, folding in each confirmed call's write-set. Two nodes with the same root at the same height provably hold the same state, and anyone can re-derive the root from the confirmed call log. Divergence is therefore detectable at a glance, and audits reduce to deterministic replay against public inputs.

6.4 Reorg Handling

SCL settlement follows the Bitcoin chain. Nodes retain undo logs and a block journal covering the last 144 blocks (approximately one day): a shallow reorganization rolls state back automatically and replays the new branch, while deeper divergence triggers a full re-derivation from the confirmed call log. Because every input is public and deterministic, both paths converge byte-identically on every honest node.

7 Data Availability and Partitioned State

An envelope call commits a *hash* of the signed payload on Bitcoin; the payload itself travels off-chain through the node network. This keeps Bitcoin usage minimal, but it creates a gap: the chain can prove that a call happened without every node holding the data required to execute it. If a payload is withheld, execution of the affected state must stall until the data arrives or the call expires. The design question that matters is the *blast radius* of that stall: if the stall unit were the whole contract, one withheld payload could freeze every holder of a token for the price of a single transaction fee.

7.1 Function Classification

SCL functions are classified as *global* or *partition-scoped*. For a partition-scoped function, the state a call can touch is derivable from its anchoring Bitcoin transaction alone: the token-bound input slots the transaction consumes, plus the output slots the transaction itself creates. This changes the stall unit from the whole contract to the withholder's own transaction footprint. If one user withholds the payload for their transfer, only the slots their transaction consumed and created freeze; every other user's calls read and write disjoint slots and execute normally.

Address-keyed balance maps are deliberately *refused* partition classification. A recipient's balance key can be touched by anyone who sends to them, so under address-keyed state a withheld transfer would freeze the recipient's balance merely for having been named as recipient, a griefing vector costing one transaction fee. Partition scoping is therefore tied to UTXO-bound state, where Bitcoin already guarantees that each outpoint is consumed exactly once.

7.2 Declarations Are Checked, Not Trusted

A classification is a performance promise, never a safety assumption. When a withheld payload finally arrives, the VM captures the call's *actual* read and write set and checks it against the partition that was frozen. A call that escaped its declared partition degrades to the standard rollback-and-replay path, which converges byte-identically on every node regardless. A wrong or malicious declaration can cost performance; it cannot corrupt state or cause divergence.

7.3 Time-Boxed Stalls

Partition stalls cannot last forever. Envelope transfers carry a void window (default 144 blocks, configurable per contract at deployment): the sender may void an unrevealed transfer within the window, and at expiry an unrevealed, unvoided transfer resolves automatically by the interface's deterministic fallback, carrying the tokens to a sender-designated output. Block distance is the only clock involved, so every node resolves it identically.

7.4 Interfaces and Compact Operations

Contracts declare conformance to versioned standard interfaces through source-level annotations: `@interface(utxo_token_v1)` at contract level, and `@partition_scoped`, `@global` (the default), and `@compact_only` classifications. Conformance is enforced twice, deliberately: at compile time by the compiler, and again at registration by the node against the emitted ABI and deploy metadata, so a hand-crafted artifact cannot claim conformance it does not have. Interface identifiers are versioned by suffix; a changed requirement is always a new identifier, never a mutation of an existing one.

The first standard interface, `utxo_token_v1`, binds token ownership to Bitcoin outpoints rather than address-keyed balances. Recipients are addressed by *vout index*: a transfer names an output of its own anchoring transaction, the tokens bind to whatever outpoint that output becomes, and ownership of the outpoint is ownership of the tokens. This inherits Bitcoin's double-spend safety and composes naturally with Lightning channels and HTLCs, which are outpoint-level constructs (Section 10).

For interface-defined operations, conforming contracts may additionally use *compact operations*: the selector and arguments (at most 43 bytes; a token transfer is roughly 10) ride directly in the anchoring transaction's `OP_RETURN`, in place of the payload hash. The consequences are significant: there is no envelope and therefore nothing to withhold, so the stall surface is zero; authorization is the Bitcoin spend itself, so replay is structurally impossible; and the operation is final at Bitcoin depth, with no void window. The trade-off is that arguments become chain-visible, so compact operations are opt-in per contract. A contract deployed `@compact_only` accepts nothing else and, as a result, can never stall.

8 Security Model & Guarantees

Assumptions: System assumes Bitcoin remains secure, since security derives from Bitcoin's consensus, finality, and availability.

Guarantees:

- Deterministic replay: identical inputs yield identical outputs.
- Objective disputes: correctness is provable from h, the protocol stamp, execution traces, and the block's state root.
- Automatic rejection: invalid transitions are discarded without forks or coordination.
- Replay and front-run resistance: anchor-outpoint binding ties every call envelope to exactly one Bitcoin transaction (Section 6).
- Divergence detection: per-block state roots make any disagreement between nodes detectable and auditable from the confirmed call log.
- Bounded stall exposure: partition-scoped state confines data-withholding to the withholder's own transaction footprint, time-boxed by the void window (Section 7).

8.1 Determinism and Reproducibility Testing

SCL aims to ensure that independent evaluators reach identical results given the same inputs (bytecode and constants, payload, Bitcoin-derived context, and prior state). To support this goal, the reference implementation is expected to maintain:

- **Golden test vectors:** a suite of canonical contracts and payload sequences with expected outputs/state roots.
- **Cross-platform determinism checks:** continuous integration runs the same vectors across supported OS/architectures and compares outputs bit-for-bit.

- **Fuzzing/property tests:** randomized payload generation and contract-state mutation to detect panics, nondeterministic behavior, or divergence across runs.

8.2 Resource Exhaustion and Denial-of-Service Surfaces

Because SCL execution occurs off-chain, the primary availability risks are resource exhaustion and unbounded growth in locally maintained data. The reference node mitigates these risks through explicit limits and operational controls:

- **Execution ceilings:** maximum instruction count, stack depth, and call depth per invocation (see Section 4).
- **Payload and request limits:** maximum accepted payload size and maximum argument serialization size per API request.
- **Rate limiting:** per-IP or per-API-key request throttling and backpressure on expensive endpoints.
- **State growth controls:** configurable retention/indexing policies for payload history, logs, and derived indexes to prevent unbounded disk growth.

These controls do not replace correctness guaranties; they restrict the worst-case cost of evaluation and storage for adversarial workloads.

8.3 Node Trust Boundaries and Verification Responsibilities

For independent verification, it must be clear which components are trusted and which are verified:

- **Bitcoin data source:** nodes obtain transaction and UTXO information from a configured backend. Regardless of the source, the nodes should verify the critical fields used for execution (e.g., tx outputs containing commitments, relevant UTXOs, and block confirmation depth).
- **Caching vs verification:** cached Bitcoin responses and derived indexes are performance optimizations; correctness-critical decisions must remain reproducible from canonical inputs.
- **Reorg handling:** nodes treat recently seen transactions as pending until a configurable confirmation threshold is reached, and additionally retain undo logs plus a 144-block journal so that shallow reorganizations roll back and replay automatically (Section 6).

8.4 Key Management and Wallet Operational Considerations

The SCL wallet (and any deployment tooling) may control Bitcoin private keys used to authorize anchoring transactions. Operational security therefore depends on standard key-management practices:

- secure key storage (hardware wallets, OS keychains, or HSMs where applicable);
- protection against malware and clipboard/transaction-substitution attacks; and
- user confirmation of transaction outputs and `OP_RETURN` contents before broadcast.

8.5 Logging, Monitoring, and Incident Response

Operators should run nodes with:

- structured logs (execution outcome, limit violations, validation failures);
- metrics on CPU, memory, disk growth, and request rates; and
- alerting for abnormal failure rates or sustained resource pressure.

Where logs may contain sensitive data (addresses, payload content), nodes should support configurable redaction and retention policies. The SCL ecosystem provides an interface which enables all the above mentioned functionality, but does not enforce and restrictions.

9 Cryptographic Guarantees

SCL leverages strong cryptographic primitives:

- **Commitments.** $h = \mathrm{H}(C)$ binds contract semantics immutably. Any modification to C changes h with overwhelming probability.
- **Collision resistance.** Given hash function H (e.g., SHA-256), the probability of $C \neq C'$ but $\mathrm{H}(C) = \mathrm{H}(C')$ is negligible ($< 2^{-128}$).
- **Authorization.** Payloads are signed by private keys associated with Bitcoin UTXOs. Verification ensures only rightful owners trigger state changes.

- **Replay protection.** Each call envelope names the unique Bitcoin outpoint its anchoring transaction must spend, binding the envelope to exactly one transaction; together with context binding (Γ includes block height and UTXO state), signatures cannot be replayed, re-anchored, or front-run.

These guarantees yield cryptographic soundness: contracts are immutable, verifiable, and objectively resolvable without coordination.

9.1 Quantum-Resistant Methodologies

While SCL relies on well-established classical cryptography, prudent system design anticipates future advances in cryptanalysis, including the advent of large-scale quantum computers. Although SCL does not enforce quantum-resistant mechanisms at the protocol level, several mitigation strategies are strongly recommended at the wallet and client layers.

9.1.1 Hierarchical Deterministic (HD) Key Usage.

Clients SHOULD employ Hierarchical Deterministic (HD) wallets (e.g., BIP-32/44-style derivation) and generate a fresh address for each authorization event. This practice minimizes public key reuse and materially reduces exposure to quantum attacks targeting elliptic curve cryptography.

On Bitcoin, a public key is not revealed on-chain until it is spent. By ensuring that each UTXO is associated with a unique public key that is used exactly once, HD key derivation limits the window during which an adversary, classical or quantum, can observe a public key and attempt to derive its corresponding private key.

Quantum Threat Model Considerations. Under a quantum threat model (e.g., Shor's algorithm applied to ECDSA), the primary risk arises *after* a public key is revealed. Address reuse amplifies this risk by exposing the same public key across multiple transactions and contexts. In contrast, single-use keys ensure that even if a quantum adversary could derive a private key from a revealed public key, the compromised key cannot be reused to authorize future state transitions.

Client-Side Enforcement. These quantum-resilient practices are not enforced by SCL consensus rules and do not alter protocol validity. However, wallet and client implementations SHOULD enforce:

- One-time-use addresses for authorization payloads,
- Deterministic key derivation to prevent entropy reuse,
- Automatic avoidance of public key reuse across distinct contract contexts.

By confining long-term cryptographic exposure to hash-based commitments (which are believed to be quantum-resistant up to quadratic speedups) and minimizing public key reuse, SCL maintains a robust security posture against both classical and foreseeable quantum adversaries without requiring immediate migration to post-quantum signature schemes.

9.1.2 Post-Quantum Authorization via Taproot: Client and Protocol Considerations

SCL does not presently mandate or implement post-quantum (PQ) cryptographic mechanisms at the protocol level. However, its design is intentionally compatible with wallet- and client-level strategies that provide quantum-resistant authorization, and it leaves clear extension points for future protocol-level integration and smart contract exposure.

Wallet-Enforced PQ Authorization. SCL-compatible wallets MAY construct Taproot outputs that commit to script paths requiring verification of a post-quantum signature (e.g., lattice-based or hash-based schemes) over a well-defined authorization message. The associated PQ public key, or a cryptographic commitment to it, is bound to the output at creation time. When this authorization path is exercised, the wallet reveals the committed script and supplies the corresponding PQ signature as witness data.

All verification is performed deterministically by Bitcoin full nodes under existing consensus rules. No trusted intermediaries, protocol modifications, or off-chain attestations are required, preserving a trustless execution model.

Optionality and Cost Management. Post-quantum signatures incur substantially larger witness sizes and higher on-chain costs. Taproot script-path semantics ensure that PQ data is revealed only when the post-quantum authorization path is exercised. This enables efficient operation under classical assumptions while preserving a quantum-resistant execution path when required.

Protocol and Smart Contract Extensibility. While PQ authorization is currently optionally enforced at the wallet level, SCL's execution and verification model does not preclude future protocol-level recognition of post-quantum authorization conditions. In such an extension, PQ-verified events could be surfaced as first-class predicates within SCL smart contracts, enabling contracts to explicitly require, observe, or branch on quantum-resistant authorization without altering underlying Bitcoin consensus rules.

10 SCL over the Lightning Network

The UTXO-bound token model of Section 7 composes directly with the Bitcoin Lightning Network, because channels and HTLCs are themselves outpoint-level constructs. SCL assets are bound to a Lightning channel at open time: the channel funding transaction is built from the opener's token-bound UTXOs, and the binding is registered against the channel's funding outpoint. From that moment, a per-channel *asset allocation*, sequence-numbered and co-signed by both parties, is the authoritative record of ownership inside the channel, maintaining the invariant that the two sides always sum to the bound total.

Asset transfers then ride *ordinary* Lightning payments. The recipient registers asset terms (contract identifier and amount) against a standard BOLT11 invoice's payment hash; intermediate hops route the payment normally, with no knowledge of the asset layer; and when the payment settles, the allocation update is applied atomically with the HTLC preimage release, keyed to the same payment hash, so the asset transfer settles exactly when the payment does. Balances therefore move instantly and off-chain, with nothing touching Bitcoin until the channel closes.

Settlement back to Bitcoin at channel close, cooperative or forced, is automated and crash-safe: detected closes are persisted as durable settlement intents before any settlement is forwarded, intents are acted upon only after a confirmation threshold and re-verification (so reorganizations roll intents back), the final split maps each party's co-signed allocation onto the closing outputs that pay them, and the settlement call on the SCL node is idempotent. Custody remains client-side throughout: a WebAssembly signer derives on-chain and per-channel keys from one seed and signs locally in the user's browser, while an always-online hub performs Lightning connectivity and channel management but can only *request* signatures, never produce them.

11 Resource Metering and the Incentive Layer

Every execution produces an *execution receipt* recording the instructions executed, peak call depth, cross-contract calls, and net storage written. One opcode is one instruction; there is no per-opcode gas table, which keeps replay deterministic and cheap for validators. Nested calls share one budget and one receipt, so a transaction's resource consumption is a single consensus-checked fact.

Receipts are the basis of the network's incentive layer, denominated in the network token XRB. Validators earn XRB from a fixed emission pool released per Bitcoin block for replaying and verifying execution; there is no block subsidy and no inflation beyond the fixed pool. Contracts stake XRB to set their execution capacity in tiers, and burn small, deterministically priced amounts only for work beyond a free envelope sized to a single basic operation, so simple operations such as token transfers remain free while complexity (computation, cross-contract composition, and bulk state growth) is what pays. Every charge is additionally scaled by a pure integer function of `total_supply`, so that cumulative burn asymptotically approaches, and can never exceed, a fixed burn budget; the same function is replayed identically by every validator, in the same class of rule as Bitcoin's halving schedule. The concrete fee schedule consists of consensus constants that activate at a protocol-defined block height; the full economic model is specified separately from this paper.

12 Performance Considerations

Because inputs and context are fixed and public, histories can be replayed offline and batched. Near-fully batched payloads can yield orders-of-magnitude throughput gains when amortized over a single Bitcoin transaction (e.g., illustrative $7 \rightarrow 70{,}000$ TPS under 100% batching). These represent theoretical upper bounds and depend on workload characteristics.

12.1 Commitment Storage.

SCL stores contract commitments as **32-byte hashes** embedded in Bitcoin transactions via `OP_RETURN` outputs. This minimal footprint ensures immutable anchoring while respecting network constraints.

The choice of `OP_RETURN` has historically been debated, particularly regarding the *data_carrier_limit*, which caps the maximum size of arbitrary data per output. By restricting commitments to 32 bytes, SCL minimizes resource usage while maintaining robust cryptographic guarantees. This approach aligns with Satoshi Nakamoto's original design principles and imposes negligible burden on the Bitcoin network, demonstrating that compact commitments can coexist safely with global consensus operations. Interface-defined compact operations (Section 7) use the same footprint budget to carry the operation itself, at most 43 bytes, in place of a hash.

13 Resource Limits

To ensure that the language is practically usable yet remains safe for a blockchain environment, the following execution constraints are imposed:

- **Termination Guarantee:** All functions must be statically analyzable to guarantee termination. Unbounded recursion and unverified while-loops are disallowed.
- **Bounded Loops:** For-loops are permitted but require statically verifiable bounds.
- **Metered Storage:** Persistent state updates are strictly bounded in size to prevent abuse.

- **Deterministic Execution:** No operation may depend on non-deterministic inputs (e.g., wall-clock time or randomness without verifiable seeds).

14 Conclusion

SCL demonstrates that deterministic, resource-bounded smart contracts can be securely anchored to Bitcoin without introducing the inefficiencies of general-purpose on-chain virtual machines. By restricting the language to statelessly verifiable primitives, bounded control flow, and explicitly defined resource ceilings, SCL guarantees termination, determinism, and replayability. Anchoring is end-to-end: contract deployments and every subsequent call are committed on Bitcoin, with anchor-outpoint binding closing replay and front-running structurally, per-block state roots making divergence detectable at a glance, and partition-scoped state confining data-withholding to the withholder's own transaction footprint. The integration with Bitcoin's UTXO model provides cryptographic immutability and global auditability, off-chain execution enables scalability far beyond traditional approaches, and outpoint-bound assets extend directly onto the Lightning Network for instant off-chain transfers that settle back to Bitcoin at channel close.

This design yields a practical trade-off: SCL intentionally forgoes Turing completeness in favor of safety, efficiency, and predictable execution. Despite these restrictions, the language remains expressive enough for the overwhelming majority of real-world applications, from conditional payments to multi-party protocols and verifiable data commitments.

Future work will focus on extending the catalogue of standard interfaces beyond `utxo_token_v1`, activating the metered fuel schedule of Section 11, formalizing a proof-carrying execution model, and deepening Lightning-native asset infrastructure.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[2] P. Todd, "Closed Seal Sets and Truth Lists for Better Privacy and Censorship Resistance," 2016. Available at: `https://petertodd.org/2016/closed-seal-sets-and-truth-lists-for-privacy`.

[3] M. Orlovsky, "Scalable Consensus for Client-side Validated Smart Contracts," 2025. Available at: `https://github.com/RGB-WG/yellowpaper/blob/4c18a6ac346a4dd379766f0bf2fcc02a3ca7e722/rgb-yellowpaper.pdf`.